Cursul 3
TIPURI DE DATE DERIVATE

	Obiective	
	Prezentare generala	
Pointeri	Referinte	Masive
Structuri	Utilizarea constructiei *typedef*	Conversii de tip
	Definirea constantelor	

OBIECTIVE _

- examina componentele ce formeaza un tip de data pointer;
- studia structurile;
- studia masivele de structuri;
- examina conversiile de tip;
- examina utilizarea constructiei **typedef;**
- examina definirea constantelor.

PREZENTARE GENERALA _

Scopul acestei lectii este acela de a prezenta utilizarea in C a pointerilor, referintelor, masivelor, conversiilor si a definirii constantelor in vederea construirii unor programe cat mai flexibile folosindu-ne de ceea ce ne ofera limbajul C si care il deosebeste fata de celelalte limbaje de programare de nivel inalt - **folosirea pointerilor si a referintelor**. Prin utilizarea acestora utilizatorul poate avea control asupra anumitor locatii de memorie, organizandu-si mai bine algoritmul de implemnetare, pentru realizarea unei cat mai mari flexibilitati si portabilitati a programului.

Pe langa acestea, limbajul C mai contine in arhitectura sa si alte tipuri derivate foarte des utilizate in scrierea programelor si anume **structurile, uniunile si masivele de structuri.** Utilizand aceste concepte proprii lui C, programatorul poate construi algoritmi foarte complecsi prin definirea unor structuri noi de date diferite de cele fundamentale.

De exemplu, odata definita o structura, aceasta va constitui un nou tip de data, putandu-se defini in continuare pointeri la acea structura, masive ale caror elemente sunt de tipul acestei structuri si, chiar mai mult, elemente de acest tip pot interveni in defiinirea acestor structuri.

POINTERI<home>

Pentru a putea defini notiunea de pointer, se impune clarificarea notiunii de variabila in limbajele de programare.

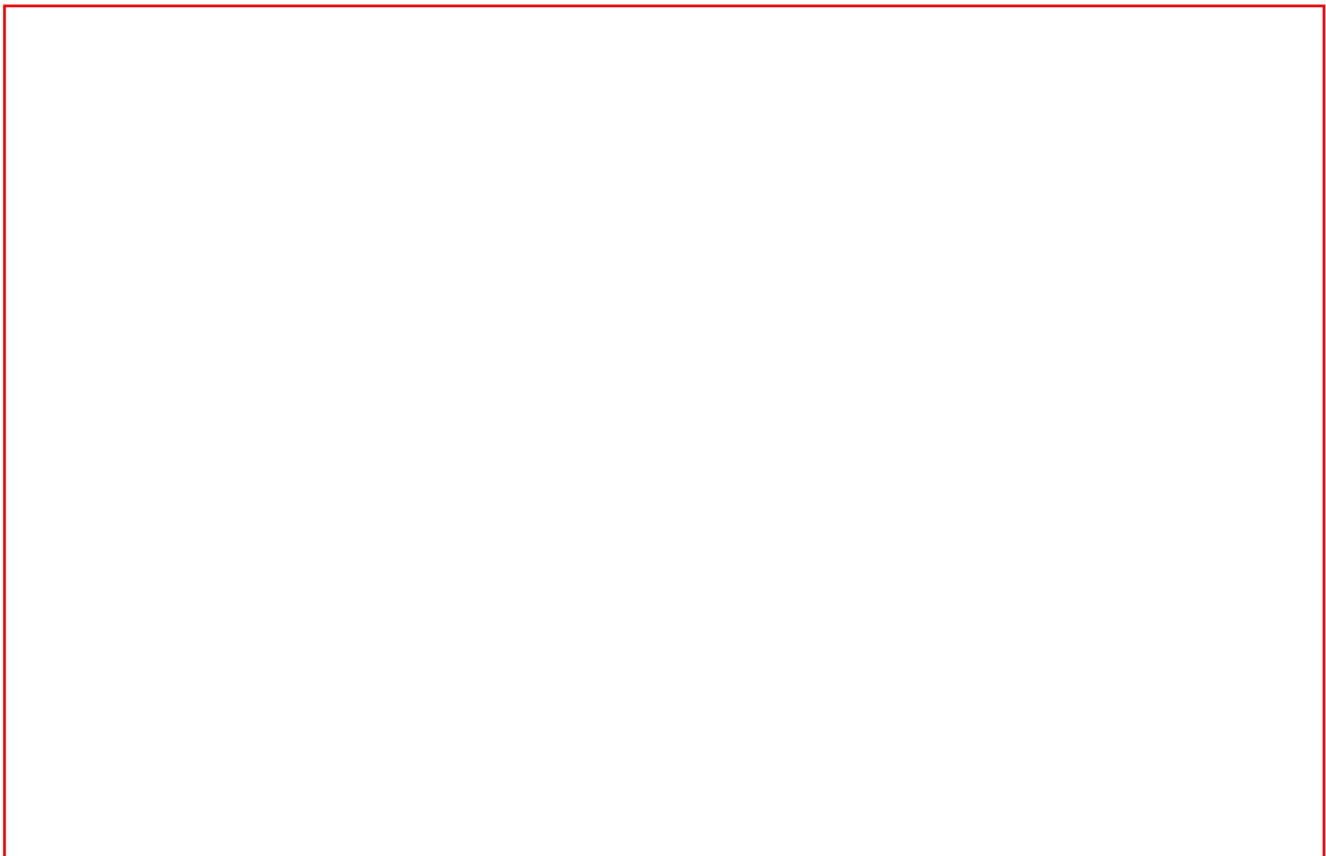
Variabila este o locatie de memorie care poate memora un obiect, ales dintr-o colectie de obiecte, manevrat in cadrul unui program. **Multimea** este domeniul de definitie al variabilei, iar locatia este o zona de memorie capabila sa memoreze orice valoare din domeniul de definitie.

Referirea la o variabila se realizeaza prin:

- utilizarea identificatorilor;
- expresiile selectoare;
- expresiile referinta.

Folosirea exacta a numelui simbolic necesita cunoasterea echivalentelor acestuia pe nivelul conceptual si cel al implementarii. Conceptual, legand notiunea de variabila de existenta unei locatii de memorie, apare o dubla ipostaza a variabilei: cea de pastratoare de date si cea de data insasi, deoarece identificarea locatiei reprezinta o informatie si implicit o data. La nivelul implementarii, unei variabile i corespunde o zona din memoria calculatorului. Conform celor relatate mai sus este evidenrt de ce unei variabile i corespunde doua valori:

- stanga (l -value), data de adresa zonei de memorie (referinta);
- dreapta (r -value), valoarea mamorata in zona respectiva (conform figurii de mai jos).



In cazul tablourilor, se va memora si un descriptor (sablon al elementelor memoratre in acesta). Asa cum se va vedea in continuare, exista situatii in care aceeasi zona de memorie poate avea mai multe nume simbolice, acesta fiind cazul structurilor **union.**

Un **pointer** este o variabila care contine adresa unei alte variabile, de orice tip. Pentru a defini un pointer, vom specifica tipul datei a carei adresa urmeaza sa o memoreze.

```
int *ip; // Pointer catre un intreg
char **s; // Pointer la un pointer pe caractere
```

Datorita acestui aspect, este clar ca, cel putin in procesul de transmitere al parametrilor unei functii, codul utilizand pointeri este mult mai compact decat codul creat fara utilizarea lor, prin intermediul pointerilor fiind posibila adresarea oricarei variabile referite de acestia. In acest context, sa consideram o variabila de tip **i** si un pointer, **pi**, catre un intreg. Cum operatorul **&** furnizeaza adresa unei variabile, instructiunea **pi=&i** asigneaza variabilei **pi** adresa lui **i** (conform figurii de mai sus). Un alt operator unar ce insoteste clasa pointerilor este *, acesta furnizand continutul locatiei de memorie de la adresa indicata de operandul sau, deci de catre un pointer. Astfel, daca **j** este tot un **int** atunci **j=*pi** asigneaza lui **j** continutul locatiiei indicate de **pi.** In acest context, are loc urmatoarea echivalenta:

```
j=*pi; <=> j=i
```

aceste atribuiri fiind precedate de :

```
int i,j;
int *pi;
pi=&i;
```

Pointerii pot aparea in expresii. De exemplu, daca **pi** indica pe **i** deci contine adresa lui **i,** atunci ***pi** poate aparea in orice context in care ar putea aparea **i,** cum ar fi :

```
j=*pi+1; // adica j=i+1;
printf("%d\n",*pi);
d=sqrt((duoble)*pi);
```

In expresii ca:

```
j=*pi+1;
```

operatorii unari * si **&** sunt prioritari fata de cei aritmetici, altfel, aceasta expresie aduna 1 si asigneaza valoarea obtinuta lui **y** ori de cate ori pointerul **pi** avanseaza.

Referiri prin pointeri pot aparea si in membrul stang al atribuirilor. Daca **pi** contine adresa lui **i** atunci ***pi=0** il pune pe **i** ca **0,** iar **pi+=1** il incrementeaza pe **i** ca si **(pi)++.**

In acest ultim exemplu, parantezele sunt necesare, fara ele expresia incrementand pe **pi** in loc sa incrementeze ceea ce indica **pi,** deoarece operatorii unari = si + sunt evaluati de la dreapta la stanga. In sfarsit, deoarece pointerii sunt variabile, ei pot fi manevrati ca orice alta variabila. Daca **pj** este un alt pointer la **int,** atunci

```
pj=pi;
```

copiaza continutul lui **pi** in **pj,** astfel ca **pj** se modifica odata cu **pi.**

Pointerii pot fi si catre elemente fara tip, **void**. Putem atribui unui pointer **void** valoarea unui pointer **non-void,** fara a fi necesara o operatie de conversie de tip, **typecast.**

```
char *cp; // Pointer catre un caracter
void *vp; // Pointer catre void
```

MASIVE<home>

Masivele de date sau tablourile, din randul carora provin vectorii si matricile, sunt tipuri de date foarte apropiate pointerilor si referintelor. Pe parcursul prezentarii se va demonstra ca orice operatie care poate fi rezolvata prin indexarea tablourilor poate fi rezolvata si cu ajutorul pointerilor.

Versiunea de rezolvare cu pointeri este mai rapida decat cea cu masive.

Tablourile sunt definite prin intermediul perechilor de paranteze " [ ]".

De exemplu: declaratia

```
char linie[80];
```

Defineste **linie** ca fiind un sir de 80 de caractere, si in acelasi timp, **linie** va constitui un pointer la caracter. Daca **pc** este un pointer la caracter, declarat prin

```
char *pc;
```

Atunci atribuirea

```
pc=&linie[0];
```

face ca **pc** sa refere primul element al tabloului **linie** (de indice 0); aceasta inseamna ca **pc** contine adresa lui **linie[0].** Acum, atribuirea

```
c=*pc;
```

va copia continutul lui **linie[0]** in **c.**

Daca **pc** indica un element oarecare a lui **linie,** atunci prin definitie, **pc+1** indica elementul urmator si, in general, **pc-i** indica cu **i** elemente inaintea elementului indicat de **pc,** iar **pc+i** cu **i** elemente dupa acelasi element. Astfel, daca **pc** indica elementul **linie[0], *(pc+1)** refera continutul lui **linie[1], pc+i** este adresa lui **linie[i],** iar ***(pc+i)** este continutul lui **linie[i].**

Aceste remarci sunt adevarate indiferent de tipul variabilelor din tabloul **linie.** Definitia adunarii unitatii la un pointer si, prin extensie, toata aritmetica pointerilor consta, de fapt, in calcularea dimensiunii memoriei ocupate de obiectul indicat. Astfel, in **pc+i, i** este inmultit cu lungimea obiectelor pe care le refera **pc** inainte de a fi adunat la **pc.**

Corespondenta intre indexare si aritmetica pointerilor este, evident, foarte stransa. De fapt, referinta la un tablou este convertita de compilator intr-un pointer spre inceputul tabloului. Efectul este ca, numele unui tablou este o expresie de tip pointer. Aceasta are cateva implicatii utile. Din moment ce **numele unui tablou este sinonim cu locatia elementului sau zero,** asignarea:

```
pc=&linie[0];
```

poate fi scrisa si

```
pc=linie;
```

Trebuie tinut seama de o diferenta ce exista intre numele unui tablou si un pointer. **Un pointer este o variabila,** astfel ca **pc=linie** si **pc++** sunt operatii permise. In schimb, **un nume de tablou este o constanta** si nu o variabila, constructii de tipul **linie=pc** sau **linie++** fiind interzise. De fapt, singurele operatii permise a fi efectuate asupra numelor masivelor, in afara celor de indexare, sunt cele care pot actiona asupra constantelor.

ARITMETICA ADRESELOR. C este consistent si constant cu aritmetica pointerilor, pointerii, tablourile si aritmetica adresarii constituind unul din punctele forte ale limbajului. C garanteaza ca nici un pointer care contine adresa unei date nu va contine valoarea 0, valoare rezervata semnalelor de eveniment anormal. De fapt aceasta valoare este atribuita constantei simbolice **NULL** pentru a indica mai clar, aceasta este o valoare speciala pentru un pointer. In general, intregii nu pot fi asignati pointerilor, zero fiind un caz special.

Exista situatii in care pointerii pot fi separati. Daca **p** si **q** indica elemente ale aceluiasi tablou, operatorii <,>,=, etc., lucreaza conform asteptarilor.

```
P<q
```

este adevarata, de exemplu in cazul in care **p** indica un element anterior elementului pe care il indica **q.** Relatiile == si != sunt si ele permise. Orice pointer poate fi testat cu **NULL** dar nu exista nici o sansa in a compara pointeri situati in tablouri diferite. Mai poate sa apara si situatia nefericita in care codul va merge pe un echipament, si sa nu functioneze pe altul.

TABLOURI MULTI-DIMENSIONALE. Este posibila definirea masivelor multidimensionale cu ajutorul tablourilor de tablouri:

```
char ecran[25] [80];
```

exceptie facand tablourile de referinte, acestea din urma nefiind permise, datorita faptului ca **nu sunt permisi pointeri la referinte**.

Tablourile sunt memorate pe linii, si deci, ultimii, de la stanga la dreapta, indici variaza mai repede decat primii. Prima dimnesiune a unui masiv se foloseste numai pentru a determina spatiul ocupat de acesta, ea nefiind luata in consideratie decat la determinarea unui element de indici dati. **Este permisa omiterea primei dimensiuni a unui tablou, daca tabloul este extern, alocarea facandu-se in cadrul altui modul, sau cand se efectueaza initializarea tablolui in declaratie, in acest ultim caz fiind determinata dimensiunea din numarul de elemente initializate.**

Initializarea masivelor poate avea loc chiar in cadrul declararii acestora:

```
int point[2]={10,19};
char mesaj1[6]={'S','a','l','u','t','\0'};
char mesaj2[6]="Salut";
```

Se observa ca sirul de caractere al lui **mesaj1** are 6 caractere avand drept terminator de sir caracterul **nul**. Diferenta intre cele doua siruri nu se afla in continutul lor, ci in cadrul initializarii lor. In cazul initializarii prin acolade, { }, caracterul nul nu este subinteles, prezenta acestuia ramanand la dispozitia utilizatorului, in schimb, folosind ghilimelele va trebui sa dimensionam corespunzator sirul de caractere, tinand cont de prezenta terminatorului de sir.

STRUCTURI<home>

Limbajul C poseda inca un tip derivat de date care este utilizat intens in elaborarea programelor si care

este foarte apreciat. Este vorba despre strctura (**struct**) care incapsuleaza unul sau mai multe elemente. Spre deosebire de masiv unde elementele sunt toate de acelasi tip, la structura elementele, denumite membrii, pot fi de tipuri diferite.

Deci, o **structura** este o colectie de date, eventual de tipuri diferite, si care pot fi referite atat separat, cat si impreuna. Definirea unei structuri se face cu cuvantul cheie **struct.**

De exemplu:

```
struct Coordinate {
Int x;
Int y;
}
```

Structura a carei denumire este **Coordinate** este formata din doi membrii, **x** si **y** fiecare de tipul intreg.

```
struct punct { float x,y;} p;
```

S-a definit **p** ca fiind de tip **punct**, punctul fiind compus din doua elemente reale **x** si **y.** Declaratia unei structuri se va termina in mod obligatoriu cu punct si virgula. Asupra elementelor unei structuri putem actiona prin intermediul operatorului de apartenenta , ".".

```
p.x=10;
p.y=30;
```

Exista posibilitatea efectuarii de opeartii cu intreaga structura, atribuirea fiind una dintre ele:

```
p={10,30};
```

O declaratie de structura care nu este urmata de o lista de variabile, nu produce alocarea memoriei, ci descrie organizarea structurii. Membrii unei strcturi pot avea tipuri diverse. Poate aparea ciudat, insa un membru al unei structuri, o eticheta sau o variabila simpla pot avea acelasi nume, fara a da ocazia unei ambiguitati. Acesta se dataoreaza operatorului ".", acesta legand numele membrului de numele structurii.

Odata definita o structura, aceasta va constitui un nou tip de data, putandu-se defini in continuare pointeri la acea structura, masive ale caror elemente sunt de tipul acestei structuri si, chiar mai mult, elemente de acest tip pot interveni in definirea altor structuri.

```
struct record {
char name[40];
```

```
char address[64];
float weight;
}
```

Primii trei membrii sunt masive, iar al patrulea este un **float.** Numele **record** este de fapt un sablon (template) pentru obiectele cu care se va lucra. Definirea propriu-zisa inca nu a fost facuta. Ca sa alocam realmente spatiu vom proceda astfel:

```
struct record rec;
```

Initializarea membrilor "variabilei" **rec** se face prin intermediul functiei **strcpy** in felul aratat in continuare:

```
.......
struct Record rec;
strcpy(rec.name, "Rozor Cristian");
...
rec.weight=145;
...
```

Un alt aspect al utilitatii structurilor il constituie tratarea tablourilor de structuri. De exemplu:

```
punct hexagon[6];
punct octogon[8];
```

Accesul catre membrii componenti ai fiecarui element al unui vector se realizeaza prin combinarea celor doua sintaxe, cea indexata, caracteristica masivelor, cu cea utilizata in cazul structurilor:

```
hexagon[i].x=10;
```

In cazul definirii unui pointer la o structura, accesul la componentele acestei structuri se va efectua prin expresii de forma:

```
punct *pptr;
pptr->x=10; //echivalent cu p.x=10;
(*pptr).y=30 //echivalent cu p.y=30;
```

parantezele neavand decat rolul de a indica ordinea in care actioneaza cei doi opeartori, "*" si ".", prioritar fiind "*".

UTILIZAREA CONSTRUCTIEI *typedef*

Un mod de a evita utilizarea cuvintelor **enum, struct** sau **union** in cadrul programelor C este de a **defini tipul de data.** Pentru aceasta vom utiliza **typedef.**

```
Typedef struct {float x,y;} punct;
```

pentru a defini noul tip de data, si apoi, il vom utiliza in declararea variabilelor.

```
Punct p;
```

Asa cum s-a vazut in exemplele anterioare, in C **typedef** nu mai este necesar in astfel de situatii, C considerand numele tipurilor de date ca fiind identificatorii utilizati in definirea agregatelor de tipuri.

Deci, in C putem scrie:

```
struct punct {float x,y;};
```

urmand ca variabilele sa fie declarate prin:

```
punct p;
```

Exista totusi situatii diferite de cele anterioare, in care instructiunea **typedef** isi dovedeste utilitatea. Acestea sunt cazurile in care dorim sa definim tipuri sinonime de date, de regula prescurtari ale tipurilor recunsocute.

Exemple:

```
typedef unsigned int ui; // Defineste tipul ui
typedef char *string; // Defineste tipul string
```

In implementarea ANSI C , pentru a defini o structura trebuie folosit cuvantul cheie **struct.** Spre exemplu presupunand ca exista sablonul **point** se poate defini obiectul **mypoint** astfel:

```
struct Point mypoint;
```

sau se mai poate scrie

```
point mypoint;
```

sau se poate recurge la constructia urmatoare pentru mai multa flexibilitate

```
typedef <type> Name;
```

adica se atribuie numelui simbolic **Name** tipul de data **type** care va putea fi apoi utilizat in locul acelui tip.

```
typedef struct tagPoint {
Int x,y;
} point;
```

Ceea ce s-a scris mai sus este o declaratie si anume: se atrbuie numelui simbolic **Point** tipul **struct.** Insa obiectul declarat trebuie sa aiba un nume. Pentru aceasta s-a ales **tagPoint.** Dupa aceasta in cadrul programului se va defini variabila **p** sau **mypoint** astfel:

```
point p,mypoint;
```

Compilatorul, in schimb va vedea de fapt o linie de cod de genul urmator:

```
struct tagPoint p,mypoint;
```

CONVERSII DE TIP

Conversiile de tip se pot realiza atat implicit, cat si explicit. Un exemplu de situatie in care are loc conversia implicita este aceea in care este asteptat un intreg si apare un caracter.. Lucrurile, in acest caz, se desfasoara fara complicatii, dar nu totdeauna avem situatii atat de simple. De aceea este de preferat a se utiliza conversiile explicite in locul celor implicite.

De exemplu:

```
int i=8,j=9;
double d;
d=(double)i; // in sintaxa C
```

Totusi, deosebirea majora intre cele doua variante nu consta in forma, ci in faptul ca C permite conversii la tipuri definite de utilizatori asemeni apelurilor de functii, ANSI C neavand acesta capacitate. Cand intr-o expresie apar operanzi de mai multe tipuri, ei se convertesc intr-un tip comun, dupa un numar restrans de reguli. In general, singurele conversii care se fac automat sunt acelea cu sens, de exemplu, convertirea unui numar intreg intr-un flotant in expresii de tipul **f+i**. Expresiile fara sens, de exemplu, folosirea lui **float** ca indice de tablou este interzisa.

In primul rand **char** si **int** pot fi amestecati in expresiile aritmetice, orice **char** fiind convertit intr-un **int.** Aceasta permite o flexibilitate remarcabila in anumite tipuri de transformari de caractere. De exemplu prezentam functia **atoi**() care converteste un sir de cifre in echivalentul lor numeric:

```
atoi (char s[]) //converteste un sir s intr-un intreg
{ int i,n=0;
for (i=0; s[i]>='0'&& s[i]<='9';++i)
n=10*n+s[i]-'0';
return n;
}
```

Expresia:

```
s[i]-'0'
```

reprezinta valoarea numerica a caracterului aflat in **s[i]** deoarece valorile lui 0, 1, etc., formeaza un sir crescator, pozitiv si continuu.

In general daca un operator binar ca + sau *, are operanzi de tipuri diferite, tipul inferior este promovat la tipul superior inaintea executiei operatiei. .Rezultatul insusi este de tipul superior. Mai precis, pentru ficare operator aritmetic se aplica urmatoarea secventa de reguli de conversie: **char** si **short** se convertsec la **int**, iar **float** este convertit la **double**, iar rezultatul este **double**. Altfel, daca un operand este **long**, celalalt este convertit in **long**, iar rezultattul este **long**. Altfel daca un operand este **unsigned**, la **unsigned** este convertit si operandul celuilalt, iar rezultatul este tot **unsigned**. Altfel, operanzii trebuie sa fie de tip **int**, iar rezultatul este un **int**.

Conversiile se fac si in asignari; valoarea membrului drept este converitita la tipul din stanga, care este tipul rezultatului. Un **char** este convertit intr-un **int** fie cu extensie de semn sau fara.

```
int i;
char c;
i=c;
c=i;
```

valoarea lui **c** este neschimbata. Acest lucru este adevarat si cand extensia de semn este implicita si cand nu este implicita.

Daca **x** este **float** si **i** este **int,** atunci:

```
x=i;
```

```
i=x;
```

provoaca amandoua conversii: **float** in **int** provoaca trunchierea oricarei parti fractiunare; **double** este convertit in **float** prin rotunjire. Intregii lungi sunt convertiti in scurti sau in **char** prin pierderea bitilor de ordin superior in exces.

Deoarece argumentul unei functii este o expresie, conversia de tip are loc si cand argumentele sunt pasate functiei; in particular, **char** si **short** devin **int,** iar **float** devine **double**. Iata de ce se va declara argumentul unei functii ca fiind **int** si **double,** chiar daca functia este apelata cu **char** si **float.** In final, convesia explicita de tip poate fi fortata in orice expresie cu o constructie numita **typecast.**

De exemplu, rutina din biblioteca, **sqrt(),** are nevoie de un argument **double** si va produce nonsens daca i se transmite altceva.. Astfel daca **n** este un intreg :

```
sqrt((double)n);
```

Sau

```
sqrt(double(n));
```

Il converteste pe **n** in **double**, inainte de a-l pasa lui **sqrt()**.

DEFINIREA CONSTANTELOR

Definirea constantelor este ca si la variabile un proces de alocare de spatiu de memorie. Insa o constanta este exact opusa unei variabile, **deoarece ea nu isi modifica valoarea** dupa cum arata si denumirea.

CONSTANTE LITERALE.

Sunt constante a caraor valoare este precizata prin introducerea directa de la tastatura. Valoarea **1234** este o constanta de tip literal din punct de vedere C. Tot asa sunt si constantele **3.14159, "Afisarea unei date…".** Exista o varietate de formate pentru constante. Valorile intregi pot de exemplu sa fie exprimate in trei coduri si anume: zecimal, hexazecimal si octal. Constanta 255 este exprimata in cele trei moduri astfel:

```
max=255; //zecimal
max=0xFF //hexazecimal
max==377 //octal
```

Intregii zecimali contin cifre cuprinse intre 0 si 9. In hexa 0 - 9 si A - F, iar in octal 0 - 7. Valorile

constantelor lungi (**long**) trebuie sa fie insotite de litera **L**. asa cum este exemplificat mai jos:

```
long population;
population=655982L;
```

Constantele in virgula mobila folosesc doua notatii, zecimala si stiintifica:

```
float pi;
pi=3.14159; //notatia zecimala
```

Notatia stiintifica se prefera in contextul valorilor foarte mari sau foarte mici. De exemplu viteza luminii (in m/s) s-ar exprima astfel in aceasta notatie:

```
float lightspeed;
lightspeed=3E+8;
```

ceea ce in notatia obisnuita (zecimala) ar corespunde valorii 30000000.0. O valoare foarte mica, spre exemplu 0,0000000043 se exprima **4.3E-9**. Constanta tip sir de caractere este intotdeauna incadrata de ghilimele.

```
printf("distanta este==%d\n",distanta);
```

Constantele de tip caracter sunt incadrate de apostrofi, ca in exemplu:

```
char ch;
ch='Q';
```

Tipul de date **char** este de fapt un intreg cu valori intre 0..255. Deoarece caracterele sunt reprezentate prin intregi, acestia pot avea semn sau nu. De exemplu:

```
signed char byte;
byte=-87;
```

CONSTANTE SIMBOLICE.

O alta modalitate mai flexibila de a defini constante ce trebuiesc folosite global este de a folosi directiva **define,** ca in exemplul urmator:

```
#define PI 3.14159
#define ANDROMEDA 130000
```

Aceste denumiri nu sunt variabile ci, numai simboluri pe care compilatorul le considera ca avand o alta semnificatie. Ele pot fi folosite oriunde acolo unde este contextual sa ne referim la literali in program. In exemplul:

```
long diametrul_galaxiei;
diametrul_galaxiei=ANDROMEDA;
```

se atribuie de fapt constanta 130000 variabilei **diametrul_galaxiei**.

Directiva **#define** are trei parti: cuvantul cheie, denumirea simbolica si valoarea. In mod traditional se obisnuieste ca denumirea sa fie mentionata cu majuscule. Directiva **#define** nu defineste *obiectul* de fapt. Ea defineste un nume simbolic pentru o valoare literala. La intalnirea denumirii respective compilatorul va inlocui (translitera) acel simbol cu literalul in sine.